KPL Syntax

Harry H. Porter III
Portland State University

HHPorter3@gmail.com

8 March 2021

This document gives syntax of the KPL programming language. KPL is the programming language of the Blitz-64 Computer System.

Table of Contents

Notation Used in the Grammar

This document provides a Context-Free Grammar (CFG) for the KPL language[1]. This grammar is meant to be exactly identical to the grammar the appendix of the document

"An Introduction to KPL: A Kernel Programming Language"

To make the grammar easier to read and understand, we use an extended CFG notation, which is described here.

Non-terminal Symbols are shown like this:

`HeaderFile` `Type` `Expr` `Statement` *etc...*

Terminal Symbols:

Keywords are shown in boldface, like this:

`if` **`while`** **`int`** **`endWhile`** *etc...*

The following lexical tokens appear in the grammar:

	Examples	
INTEGER	**`42`**	**`0x1234ABCD`**
DOUBLE	**`3.1415`**	**`6.022e23`**
CHAR	**`'a'`**	**`'\n'`**
STRING	**`"hello"`**	**`"\t\n"`**
ID	**`myName`**	**`MAX_SIZE`**
OPERATOR	**`<=`** **`<`** **`>`** **`>=`** **`!=`** **`+`** **`-`** **`*`**	*etc...*

Punctuation Symbols

The following characters are particularly important in KPL's grammar:

`{` **`}`** **`[`** **`]`** **`|`** **`:`** **`,`** **`.`** **`=`** **`(`** **`)`** **`;`**

Of these, the following punctuation symbols conflict with grammar meta-symbols:

`{` **`}`** **`[`** **`]`** **`|`**

When used as grammar meta-symbols, they are shown without quotes:

`{` `}` `[` `]` `|`

When used as terminals, i.e., when meant literally, they are quoted:

`'{'` **`'}'`** **`'['`** **`']'`** **`'|'`**

The remaining punctuation symbols are only used as terminals and are not quoted:

`:` **`,`** **`.`** **`=`** **`(`** **`)`** **`;`**

[1] Technically, the KPL grammar is "LL(k)" and in most cases can be parsed with only a single token look-ahead, making it much easier for the human than "LR(k)" grammars.

Comments are not included in this grammar. There are two forms of commenting:

`--` *through end-of-line*

`/*` *through* `*/`

Meta-Symbols, which are used in describing the grammar:

Grammar rules: `-->`

Example:

```
Type --> int
```

Rules with alternatives are shown like this:

Example:

```
Statement --> IfStmt | AssignStmt
```

Example:

```
Statement --> IfStmt
          --> AssignStmt
```

Optional material: `[ ]`

Example:

```
VarDecl --> Decl [ = Expr2 ]
```

Repetition of zero-or-more: `{ }`

Example:

```
StmtList --> { Statement }
```

Repetition of one-or-more occurrences: `{ }+`

Example:

```
VarDecls --> var { VarDecl }+
```

A Context Free Grammar of KPL

```
HeaderFile      --> header ID
                      [ Uses ]
                      { Constants |
                        Errors |
                        VarDecls |
                        Enum |
                        TypeDefs |
                        FunctionProtos |
                        Interface |
                        Class }
                    endHeader
CodeFile        --> code ID
                      { Constants |
                        Errors |
                        VarDecls |
                        Enum |
                        TypeDefs |
                        Function |
                        Interface |
                        Class |
                        Behavior }
                    endCode
Interface       --> interface ID [ TypeParms ]
                      [ extends TypeList ]
                      [ messages { MethProto }+ ]
                    endInterface
Class           --> class ID [ TypeParms ]
                      [ implements TypeList ]
                      [ superclass NamedType ]
                      [ fields { Decl }+ ]
                      [ methods { MethProto }+ ]
                    endClass
Behavior        --> behavior ID
                      { Method }
                    endBehavior
Uses            --> uses OtherPackage { , OtherPackage }
OtherPackage    --> ID     [ renaming Rename { , Rename } ]
                --> STRING [ renaming Rename { , Rename } ]
Rename          --> ID to ID
TypeParms       --> '[' ID : Type { , ID : Type } ']'
Constants       --> const { ID = Expr }+
Decl            --> ID { , ID } : Type
VarDecl         --> Decl [ = Expr2 ]
VarDecls        --> var { VarDecl }+
Errors          --> errors { ID ParmList }+
TypeDefs        --> type { ID = Type }+
Enum            --> enum ID [ = Expr ] { , ID }
IdList          --> ID { , ID }
ArgList         --> ( )
                --> ( Expr { , Expr } )
```

```
ParmList          --> ( )
                  --> ( Decl { , Decl } )
FunctionProtos    --> functions { FunProto }+
FunProto          --> [ external ] ID ParmList [ returns Type ] [ StackUsage ]
Function          --> function ID ParmList [ returns Type ] [ StackUsage ]
                        [ VarDecls ]
                        StmtList
                      endFunction
StackUsage        --> '[' Max_Stack_Usage = Expr ']'
NamelessFunction  --> function    ParmList [ returns Type ]
                        [ VarDecls ]
                        StmtList
                      endFunction
MethProto         --> ID ParmList [ returns Type ] [ StackUsage ]
                  --> infix OPERATOR ( ID : Type ) returns Type
                  --> prefix OPERATOR ( ) returns Type
                  --> { ID : ( ID : Type ) }+ [ returns Type ]
Method            --> method MethProto
                        [ VarDecls ]
                        StmtList
                      endMethod
StmtList          --> { Statement }
Statement         --> if Expr StmtList
                          { elseIf Expr StmtList }
                          [ else StmtList ]
                      endIf
                  --> LValue = Expr
                  --> LValue += Expr
                  --> LValue -= Expr
                  --> ID ArgList
                  --> Expr { ID : Expr }+
                  --> Expr . ID ArgList
                  --> while Expr
                          StmtList
                      endWhile
                  --> do
                          StmtList
                      until Expr
                  --> break
                  --> continue
                  --> return [ Expr ]
                  --> for LValue = Expr to Expr [ by Expr ]
                          StmtList
                      endFor
                  --> for ( StmtList ; [ Expr ] ; StmtList )
                          StmtList
                      endFor
                  --> switch Expr
                          { case Expr : StmtList }
                          [ default : StmtList ]
                      endSwitch
```

```
                --> switchOnClass Expr
                        { case Expr : StmtList }
                        [ default : StmtList ]
                    endSwitchOnClass
                --> try StmtList
                        { catch ID ParmList : StmtList }+
                    endTry
                --> throw ID ArgList
                --> free Expr
                --> debug [ STRING ]
                --> printf ( [ ID , ] STRING { , Expr } )
                --> sprintf ( ID , STRING { , Expr } )
                --> initializeArray ( Expr )
                --> setArraySize ( Expr , Expr )
Type            --> byte
                --> halfword
                --> word
                --> int
                --> double
                --> bool
                --> void
                --> typeOfNull
                --> anyType
                --> ptr to Type
                --> struct { Decl }+ endStruct
                --> union { Decl }+ endUnion
                --> array [ '[' Dimension { , Dimension } ']' ] of Type
                --> function ( [ Type { , Type } ] )
                        [ returns Type ] [ StackUsage ]
                --> NamedType
NamedType       --> ID [ '[' Type { , Type } ']' ]
TypeList        --> NamedType { , NamedType }
Dimension       --> * | Expr
Constructor     --> Type ClassStructInit
                --> Type ArrayInit
                --> Type
ClassStructInit --> ID '{' ID = Expr { , ID = Expr } '}'
ArrayInit       --> ID '{' [ Expr of ] Expr
                          { , [ Expr of ] Expr } '}'
LValue          --> Expr
Expr            --> Expr2  {   ID : Expr2 }
Expr2           --> Expr3  {   OPERATOR Expr3 }
Expr3           --> Expr5  {   '||' Expr5 }
Expr5           --> Expr6  {   && Expr6 }
Expr6           --> Expr7  {   '|' Expr7 }
Expr7           --> Expr8  {   ^ Expr8 }
Expr8           --> Expr9  {   & Expr9 }
Expr9           --> Expr10 {   == Expr10
                             | != Expr10 }
Expr10          --> Expr11 {   < Expr11
                             | <= Expr11
                             | > Expr11
                             | >= Expr11 }
```

```
Expr11          --> Expr12 {   << Expr12
                             | >> Expr12
                             | <<< Expr12
                             | >>> Expr12 }
Expr12          --> Expr13 {   + Expr13
                             | - Expr13 }
Expr13          --> Expr15 {   * Expr15
                             | / Expr15
                             | % Expr15 }
Expr15          --> OPERATOR Expr15
                --> Expr16
Expr16          --> Expr17 {   . ID ArgList
                             | . ID
                             | '[' Expr { , Expr } ']' }
Expr17          --> ( Expr )
                --> null
                --> true
                --> false
                --> self
                --> super
                --> INTEGER
                --> DOUBLE
                --> CHAR
                --> STRING
                --> NamelessFunction
                --> ID
                --> ID ArgList
                --> new Constructor
                --> alloc Constructor
                --> sizeOf ( Type )
                --> asPtrTo ( Expr , Type )
                --> asInteger ( Expr )
                --> arraySize ( Expr )
                --> arrayMaxSize ( Expr )
                --> isInstanceOf ( Expr , Type )
                --> isKindOf ( Expr , Type )
```

Expressions

Here is simplified grammar for expressions. This rule ignores:

- Precedence
- Associativity

```
Expr --> Expr BinaryOperator Expr
     --> UnaryOperator Expr
     --> Expr . ID ( …Arguments… )
     --> Expr . ID
     --> Expr '[' Expr { , Expr } ‘]'
     --> * Expr
     --> & Expr
     --> null | true | false | nan | inf | self | super
     --> ID
     --> INTEGER
     --> DOUBLE
     --> CHAR
     --> STRING
     --> function ( …Arguments… ) ... endFunction
     --> new   Type [ '{' …Initialization… ‘}' ]
     --> alloc Type [ ‘{' …Initialization… '}' ]
     --> sizeOf ( Type )
     --> asPtrTo ( Expr , Type )
     --> asInteger ( Expr )
     --> arraySize ( Expr )
     --> arrayMaxSize ( Expr )
     --> isInstanceOf ( Expr , Type )
     --> isKindOf ( Expr , Type )
     --> ID ( ArgList )
     --> ( Expr )

BinaryOperator --> + | - | * | / | % | >> | << | >>> | <<< |
                   < | > | <= | >= | == | != | & | '|' | ^ |
                   && | '||' | …user defined infix operators…

UnaryOperator --> ! | + | - | …user defined prefix operators…
```

Precedence of Operators

The formal syntax of KPL imposes the following precedences on these expression operators. Each operator within a group is at the same precedence level and is parsed with left associativity.

This is the same as in C, C++, and Java.

Lowest Precedence

All keyword messages, e.g., x at:y put:z
All infix operators not mentioned below
|| Short-circuit for **bool** operands
&& Short-circuit for **bool** operands
| Bitwise OR for **int** operands
^ Bitwise XOR for **int** operands
& Bitwise AND for **int** operands
== Can compare basic types, pointers, and != objects, but not structs, unions or arrays
< Can compare **byte**, **halfword**, **word**, **int**, <= **double**, and **ptr** operands > >=
<< Shift int operand left logical >> Shift int operand right logical <<< Shift int operand left arithmetic >>> Shift int operand right arithmetic
+ Can also add **ptr**+**int** - Can also subtract **ptr**-**int** and **ptr**-**ptr**
* / % Modulo operator for **int**s
Prefix - For **int** and **double** operands Prefix + For **int** and **double** operands (nop) Prefix ! For **int** and **bool** operands Prefix * Pointer dereference Prefix & Address-of All other prefix methods

.	Message Sending: x.foo(y,z)
.	Field Accessing: x.name
[]	Array Accessing: a[i,j]
()	Parenthesized expressions: x*(y+z)
constants	e.g., 123, "hello", 34.998e-23
keywords	e.g., **true**, **false**, **null**, **self**, **super**
nameless funct	e.g., **function** (...) ... **endFunction**
variables	e.g., x
function call	e.g., foo(4)
built-ins	e.g., **forceToDouble** (4)
function	e.g., **function** (...) ... **endFunction**
new	e.g., **new** Person { name="smith" }
alloc	e.g., **alloc** Person { name="smith" }
sizeOf	e.g., **sizeOf** (Person) ... in bytes
asPtrTo	e.g., **asPtrTo** (i, **double**)
asInteger	e.g., **asInteger** (ptr)
arraySize	e.g., **arraySize** (array/arrayPtr)
arrayMaxSize	e.g., **arrayMaxSize** (array/arrayPtr)
isInstanceOf	e.g., **isInstanceOf** (p,*ClassName*)
isKindOf	e.g., **isKindOf** (p,*ClassOrInterfaceName*)

Highest Precedence

Lexical Matters

Greater detail about the lexical tokens is given in the KPL Reference Manual. Here is a summary.

Comments and White Space

KPL supports two comments styles.

First, a comment may begin with /* and end with */.

Second, everything after two hyphens through end-of-line is a comment.

```
x = y + 2   -- Adjust y a little
```

Both styles may be nested.

```
/* Disable this code...
   x = y + 2   /* Adjust y a little */
*/

-- Disable this code...
--   x = y + 2   -- Adjust y a little
```

White space is defined as a sequence of one or more of:

```
Space
Tab
Newline
```

Identifiers

An ID is a sequence of letters, digits, and underscores. It must begin with a letter. Only ASCII characters are allowed. Case is significant.

Integers

An INTEGER can be expressed either in decimal or in hex:

```
12345
0x01b5f3b
```

The underscore can be used as a separator to increase readability. It is ignored:

```
12_345
0x01b_5f3b
```

Floating Point Constants

A DOUBLE number must contain either a decimal point or the “e” for the exponent.

```
123.0
123e-45
```

For both INTEGERs and DOUBLEs, a leading minus/negative sign “-“ will be parsed as a separate token and used to form an expression, such as -(1). The compiler will evaluate such expressions at compile-time, so effectively any INTEGER or DOUBLE may be negated.

```
-1        -- Preferred
-(1)      -- Equivalent
```

The underscore can be used as a separator to increase readability. It is ignored:

```
1.000_000_007
```

Character Constants

A CHAR constant is enclosed in single quotes. Any Unicode character may be included or an escape sequence may be used.

```
'A'
'∉'       -- Unicode
'\n'      -- Escape sequence
```

String Constants

A string constant consists of a sequence of zero of more characters enclosed in double quotes:

```
"Hello, world\n"
```

A string constant is represented as an array of bytes. The UTF-8 encoding scheme is used to represent the string, which may contain arbitrary Unicode characters.

```
"π ≈ 3.14"
"\U0001d70b \u2248 3.14"  -- Equivalent, with codepoints in hex
"\xf0\x9d\x9c\x8b \xe2\x89\x88 3.14"  -- Equivalent, in UTF-8
```

Escape Sequences

Here are the escape sequences that may be used in character and string constants.

	Hex	Decimal		ASCII	Code Name
\0	00	0	control-@	NUL	null
\a	07	7	control-G	BEL	alert
\b	08	8	control-H	BS	backspace
\t	09	9	control-I	HT	tab
\n	0A	10	control-J	NL/LF	newline/linefeed
\v	0B	11	control-K	VT	vertical tab
\f	0C	12	control-L	FF	form feed
\r	0D	13	control-M	CR	return
\e	1B	27	control-[	ESC	escape
\d	7f	127		DEL	delete
\"	22	34		"	double quote
\'	27	39		'	single quote
\\	5C	92		\	backslash
\xHH	HH	< any hex value >			

Keyword List

Here are the keywords used in the KPL grammar. The built-in function names are not included.

```
alloc
anyType
array
arrayMaxSize
arraySize
asInteger
asPtrTo
behavior
bool
break
by
byte
case
catch
class
code
const
continue
debug
default
do
double
else
elseIf
endBehavior
endClass
endCode
endFor
endFunction
endHeader
endIf
endInterface
endMethod
endStruct
endSwitch
endSwitchOnClass
endTry
endUnion
endWhile
enum
errors
extends
external
false
fields
for
free
function
functions
halfword
header
if
implements
inf
infix
initializeArray
int
interface
isInstanceOf
isKindOf
Max_Stack_Usage
messages
method
methods
nan
new
null
of
prefix
printf
ptr
renaming
return
returns
self
setArraySize
sizeOf
sprintf
struct
super
superclass
switch
switchOnClass
throw
to
true
try
type
typeOfNull
union
until
uses
var
void
while
word
```

About This Document

Document Revision History

Version numbers are not used to identify revisions to this document. Instead the date and the author's name are used. The document history is:

Date	Author
21 October 2019	Harry H. Porter III <document created>
18 February 2020	Harry H. Porter III <initial version completed>
8 March 2021	Harry H. Porter III <current version>

Permission to Copy

About the Author

PProfessor Harry H. Porter III teaches in the Department of Computer Science at Portland State University. He has produced several video courses, notably on the Theory of Computation. Recently he built a complete computer using the relay technology of the 1940s, which has eight general purpose 8 bit registers, a 16 bit program counter, and a complete instruction set, all housed in mahogany cabinets as shown. His technical focus and research interests have included AI and neural networks; parsing and natural language processing; logic, object-oriented, and functional programming; compilers, operating systems, interpreters, and system software; and discrete math and computational theory. He has programmed in many high-level languages and written assembly code for a variety of machines, dating back to the IBM 360/67 and Intel 8080.

Porter lives in Portland, Oregon. When not trying to figure out how his computer actually works, he skis, hikes, travels, and spends time with his children building things.

Porter holds an Sc.B. from Brown University and a Ph.D. from the Oregon Graduate Center.